4.5 非同期処理： asyncio

asyncio ライブラリは効率の良い並行処理を実現するものである．asyncio は多くの機能を提供するが，本書ではその内の**高レベル API** に関する基本的な事柄について解説する．

asyncio による並行処理の考え方は，先の「4.4 マルチスレッドプログラミング」（p.237）で解説したものとは異なり，**コルーチン**（async キーワードで宣言された関数）や**タスク**（Task オブジェクト）といった実行の単位（Awaitable **オブジェクト**）を独特の**イベントループ**で管理するものである．具体的には，複数の Awaitable オブジェクトをイベントループに登録し，それらを順番に実行する．そして登録された全ての Awaitable オブジェクトが終了するまでイベントループを実行する．各タスクの切り替えは await の式で制御する．

厳密には asyncio におけるタスクの実行は並行実行ではなく，async 宣言された Awaitable オブジェクトを，1 つのスレッド上で await 式によって実行し，それらの一時停止と切替えを管理するものである．特に，ディスクに対する入出力処理や通信といった待ち時間が多く発生する処理を管理する場合にこの実行形態が有利に働くことがある．

4.5.1 コルーチン

コルーチンとは asyncio の並行処理におけるプログラムの単位であり，async キーワードで宣言された関数として実装することができる．（asyncio ライブラリを読み込んでいない状態でも async 宣言は可能である）

例. コルーチンの定義の例

```
>>> async def msg(s):  Enter      ←コルーチン定義の開始
...       print(s)    Enter
...       return ’戻り値’   Enter
...   Enter          ←コルーチン定義の終了
```

このようにして定義された msg は通常の関数とは異なり，呼び出すと**コルーチンオブジェクト**を返す．（次の例参照）

例. コルーチンオブジェクトの生成（先の例の続き）

```
>>> c = msg(’コルーチン’)  Enter      ←コルーチンを呼び出す
>>> print( c )  Enter                  ←内容確認
<coroutine object msg at 0x000002423163F1D0>     ←コルーチンオブジェクト
```

この例の実行においては msg に定義されたプログラムは動作しておらず，得られたコルーチンオブジェクトをイベントループ上で実行するには次の例のように asyncio の run 関数を使用する．

例. コルーチンの実行（先の例の続き）

```
>>> import asyncio  Enter      ←ライブラリの読込み
>>> r = asyncio.run( c )  Enter      ←イベントループを起動してコルーチンを実行
コルーチン                          ←コルーチンからの出力
>>> print( r )  Enter               ←戻り値の確認
戻り値                              ←戻り値が得られている
```

ただし，このような方法では通常の関数を実行する場合と結果的に変わらない．これに関しては次のサンプルプログラム asyncio01.py の実行によって確認できる．

プログラム：asyncio01.py

```
 1 # coding: utf-8
 2 import asyncio           # ライブラリの読込み
 3
 4 async def msg(s):        # コルーチン：引数に与えた値を3回出力する
 5     for i in range(3):
 6         print(s)
 7     return s             # 受け取った引数を返す
 8
 9 r1 = asyncio.run( msg(’コルーチン1’) )  # この順番でイベントループを
10 r2 = asyncio.run( msg(’コルーチン2’) )  # 起動してコルーチンを
11 r3 = asyncio.run( msg(’コルーチン3’) )  # 実行する
12
13 print( ’---\n’,r1,r2,r3 )
```

このプログラムを実行すると次のような出力となる．

出力）

```
コルーチン 1
コルーチン 1
コルーチン 1                    ← 1 つ目のコルーチンが終了した後で
コルーチン 2                    ← 2 つ目のコルーチンが起動する
コルーチン 2
コルーチン 2                    ← 2 つ目のコルーチンが終了した後で，
コルーチン 3                    ← 3 つ目のコルーチンが起動する
コルーチン 3
コルーチン 3                    ← 3 つ目のコルーチンが終了
—
コルーチン 1 コルーチン 2 コルーチン 3      ←それぞれの戻り値を出力
```

この実行例からわかるように，3 つのコルーチンが順番に実行されており，並行処理とはなっていないことが確認できる．このような記述では run によって 3 つのイベントループが順番に起動，終了され，それぞれのイベントループ内でコルーチンが実行されるので並行処理のような形にはならない．

4.5.2 コルーチンとタスクの違い

1 つのイベントループ上で複数の Awaitable オブジェクトの実行を管理するには，それらを**タスク**の形にしなければならない．すなわち，コルーチン自体にはイベントループ上での実行に関する管理の機能がなく，イベントループ上でのスケジューリングの対象とするには，そのための機能を持った**タスク**にする必要がある．タスクを作成するには create_task 関数が使用できる．

書き方： create_task(コルーチン)

「コルーチン」からタスクを作成して返す．

4.5.2.1 複数のタスクの並行実行

複数のタスクを 1 つのイベントループで並行実行するには，それら複数のタスクを **await 式**で起動するためのコルーチンを 1 つ実装して，それを run によって起動するとよい．

次に，await 式によるタスクの実行開始，一時停止，切り替えについて説明する．

4.5.3 await によるタスクの一時停止と切替え

asyncio の sleep 関数を使用すると，実行中の Awaitable オブジェクトの実行を一時停止して，実行制御をイベントループ上の実行可能な次の Awaitable オブジェクトに移すことができる．このことをサンプルプログラム asyncio02.py の実行を通して解説する．

プログラム：asyncio02.py

```
1  # coding: utf-8
2  import asyncio          # ライブラリの読込み
3
4  async def msg(s):       # コルーチン：引数に与えた値を3回出力する
5      for i in range(3):
6          print(s)
7          await asyncio.sleep(1)  # ←このコルーチンを一時停止して
8      return s                    #   次のタスクに制御を移す
9
10 async def exe():    # タスクを投入して実行するコルーチン
11     t1 = asyncio.create_task( msg('コルーチン1') )  # 3つのコルーチンを
12     t2 = asyncio.create_task( msg('コルーチン2') )  # それぞれタスクに
13     t3 = asyncio.create_task( msg('コルーチン3') )  # している．
14     await t1;   await t2;   await t3                # 順番に登録して実行
15
16 asyncio.run( exe() )    # イベントループの起動
```

解説）

4〜8 行目でコルーチン msg が定義されている．このコルーチンは受け取った引数を 3 回出力するものであるが，毎回の出力の直後に asyncio.sleep(1) を await 式によって実行している．これにより，当該コルーチンは実行が一時

停止され，1 秒間に渡って休止状態となる．同時に，イベントループの上にある次の実行可能な Awaitable オブジェクトに実行の制御が移る．

10〜14 行目に定義されているコルーチン exe は msg を 3 つのタスクにして（11〜13 行目）イベントループに登録して（14 行目）実行を開始するものである．この際，create_task 関数によってコルーチン msg を**タスクオブジェクト**（asyncio.Task クラス）にしている．タスクオブジェクトは生成直後は実行されず，14 行目にある await 式によってイベントループに登録されて実行される．

このプログラムの実行結果の出力を次に示す．

出力）

```
コルーチン 1
コルーチン 2
コルーチン 3        ←ここで 1 秒弱停止
コルーチン 1
コルーチン 2
コルーチン 3        ←ここで 1 秒弱停止
コルーチン 1
コルーチン 2
コルーチン 3        ←ここで 1 秒弱停止した後，プログラムが終了
```

この実行例からわかるように，3 つのタスクが同時に実行されているように見える．

注意）次の asyncio02-2.py のように実装すると並行実行のようにはならないことに注意すること．

プログラム：asyncio02-2.py（良くない実装）

```
 1 # coding: utf-8
 2 import asyncio           # ライブラリの読込み
 3
 4 async def msg(s):        # コルーチン：引数に与えた値を3回出力する
 5     for i in range(3):
 6         print(s)
 7         await asyncio.sleep(1)  # ←このコルーチンを一時停止して
 8     return s                    #   次のタスクに制御を移す
 9
10 async def exe():    # タスクを投入して実行するコルーチン
11     t1 = asyncio.create_task( msg('コルーチン1') ); await t1    # タスクの生成と
12     t2 = asyncio.create_task( msg('コルーチン2') ); await t2    # それらの実行を
13     t3 = asyncio.create_task( msg('コルーチン3') ); await t3    # この方法で試みる
14
15 asyncio.run( exe() )    # イベントループの起動
```

このプログラムの実行結果の出力を次に示す．

出力）

```
コルーチン 1        ←ここで約 1 秒停止
コルーチン 1        ←ここで約 1 秒停止
コルーチン 1        ←ここで約 1 秒停止して 1 つ目のコルーチンが終了
コルーチン 2        ←ここで約 1 秒停止
コルーチン 2        ←ここで約 1 秒停止
コルーチン 2        ←ここで約 1 秒停止して 2 つ目のコルーチンが終了
コルーチン 3        ←ここで約 1 秒停止
コルーチン 3        ←ここで約 1 秒停止
コルーチン 3        ←ここで約 1 秒停止した後，プログラムが終了
```

各タスクが並行実行のようにはならず，順番に実行されていることがわかる．複数の非同期処理を並行処理のように実行するには，exe の末尾にまとめて await 式で実行する必要がある．

4.5.4 タスク登録の簡便な方法

asyncio の gather 関数を使用すると，タスクの生成と登録を更に簡単な形で実現できる．

書き方：　gather(Awaitable オブジェクト 1, Awaitable オブジェクト 2, …)

引数に与えた Awaitable オブジェクト（コルーチン，タスクなど）をその順序でタスクとしてイベントループに登録して実行する．

gather 関数を用いて先のプログラム asyncio02.py と同様の処理を実現するプログラム asyncio03.py を示す．

プログラム：asyncio03.py

```
1  # coding: utf-8
2  import asyncio          # ライブラリの読込み
3
4  async def msg(s):       # コルーチン：引数に与えた値を3回出力する
5      for i in range(3):
6          print(s)
7          await asyncio.sleep(1)  # ←このコルーチンを一時停止して
8      return s                    #   次のタスクに制御を移す
9
10 async def exe():    # タスクを投入して実行するコルーチン
11     g = await asyncio.gather(   # 実際にタスクを実行する
12         msg('コルーチン1'),
13         msg('コルーチン2'),
14         msg('コルーチン3')
15     )
16     print(g)    # 戻り値の確認
17
18 asyncio.run( exe() )    # イベントループの起動
```

gather 関数の戻り値がリストとして変数 g に得られ（11〜15 行目）各コルーチンからの戻り値が要素となる．

4.5.5　イベントループと run 関数

イベントループは 1 つのスレッド内に 1 つだけ実行できるものである．従って，一度 run 関数で非同期処理が開始された後は，そのイベントループが終了するまで次の run 関数は実行できない．先に示したサンプルプログラム asyncio02.py〜asyncio03.py において，exe を最上位のコルーチンとして run 関数で実行し，exe の内部で複数のタスクを登録して実行するという形式をとっているのはそのような事情による．

4.5.5.1　スレッド毎に実行されるイベントループ

イベントループは通常，run 関数の実行によって自動的に作成されるが，asyncio の new_event_loop 関数によって明示的に作成（ProactorEventLoop オブジェクトを作成）することもできる．また，イベントループは異なるスレッド毎に作成することもできるので，スレッド毎に異なるイベントループを同時に実行することもできる．このことを次のサンプルプログラム asyncio04.py で示す．

プログラム：asyncio04.py

```
1  # coding: utf-8
2  import asyncio
3  import threading
4
5  async def msg(s):       # コルーチン：引数に与えた値を3回出力する
6      for i in range(2):
7          print(s)
8          await asyncio.sleep(1)  # ←このコルーチンを一時停止して
9      return s                    #   次のタスクに制御を移す
10
11 def startLoop(eloop,rn):    # スレッド内でイベントループを開始する
12     eloop.run_until_complete( msg(rn) )
13
14 eloopA = asyncio.new_event_loop()   # イベントループ(1)
15 eloopB = asyncio.new_event_loop()   # イベントループ(2)
16
17 # スレッドを2つ生成して各々でイベント管理する
18 tA = threading.Thread( target=startLoop, args=(eloopA,'コルーチン：スレッドA') )
19 tB = threading.Thread( target=startLoop, args=(eloopB,'コルーチン：スレッドB') )
20 # スレッドの開始と終了待ち
21 tA.start();     tB.start()
22 tA.join();      tB.join()
```

このプログラムの 14,15 行目で new_event_loop 関数によって別々のイベントループを作成している．またそれらを 18,19 行目でスレッド tA，tB に与えている．スレッドとして起動する関数 startLoop はイベントループと文字列を受け取り，run_until_complete メソッドによってイベントループを起動する．

書き方：　イベントループ.run_until_complete(コルーチン)

「コルーチン」に非同期処理の最上位のコルーチンを与える．このメソッドは非同期処理が終了した際にコルーチンの戻り値を返す．run_until_complete は，asyncio.run と同様の処理を，指定したイベントループに対して実行する際に用いる．

このプログラムをスクリプトとして実行した様子を次に示す．

例. asyncio04.py の実行

```
コルーチン：スレッドA
コルーチン：スレッドB
コルーチン：スレッドA
コルーチン：スレッドB
```

2 つのイベントループが平行して動作していることがわかる．

勿論であるが，各々のスレッド内で run 関数によってイベントループを自動的に作成することもできる．次の asyncio04-2.py は更に簡潔な記述で先のプログラムと同等の処理を行うものである．

プログラム： asyncio04-2.py

```
 1 # coding: utf-8
 2 import asyncio
 3 import threading
 4
 5 async def msg(s):        # コルーチン：引数に与えた値を3回出力する
 6     for i in range(2):
 7         print(s)
 8         await asyncio.sleep(1)  # ←このコルーチンを一時停止して
 9     return s                    #   次のタスクに制御を移す
10
11 def startLoop(rn):  # スレッド内でイベントループを生成して開始する
12     asyncio.run( msg(rn) )
13
14 # スレッドを2つ生成して各々でイベント管理する
15 tA = threading.Thread( target=startLoop, args=('コルーチン：スレッドA',) )
16 tB = threading.Thread( target=startLoop, args=('コルーチン：スレッドB',) )
17 # スレッドの開始と終了待ち
18 tA.start();      tB.start()
19 tA.join();       tB.join()
```

4.5.5.2　実行中のイベントループを調べる方法

get_running_loop 関数を用いると，現行のスレッドで実行中のイベントループを調べることができる．

書き方：　asyncio.get_running_loop( )

この関数は，イベントループオブジェクト（ProactorEventLoop オブジェクト）を返す．実行中のイベントループが存在しない場合は例外（RuntimeError）が発生する．

参考）

get_running_loop 関数と類似のものとして get_event_loop も存在するが，これは将来の Python の版において廃止されることが予定されている．

4.5.6　理解を深めるためのサンプル

時間かかる処理を非同期処理（async，await）で管理する方法について考える．

例. 時間のかかる処理

```
>>> import random  Enter
>>> while True:  Enter
...     r = random.random()  Enter     ←乱数を生成し
...     if r < 2**(-26):  Enter          それが 2^-26 未満であれば
...         break  Enter                 終了するループ
...  Enter     ←反復処理の記述の終了
```

このプログラムは非常に小さな乱数（発生確率の非常に低い乱数）が得られるまで乱数生成を繰り返すもので，終了まで時間がかかる例である．（実際に試されたい）このような処理を複数起動して並行実行するサンプルを次の asyncio07.py

に示す.

プログラム：asyncio07.py

```
# coding: utf-8
import asyncio
import random
import time

# v未満の乱数が出るまで試行を続けるコルーチン
async def long_task( n, v ):
    while True:                  # 非常に時間のかかる処理
        r = random.random()      #             |
        if r < v:                #             |
            break                #             ↓
        await asyncio.sleep(0)  # イベントループに制御を移すために必要
    print( n, r )

# 上記処理を繰り返すコルーチン
async def asyncIteration( n, v, i ):
    for x in range(i):
        await long_task( n, v ) # 各回の処理の完了を待つ

# イベントループに渡す最上位のコルーチン
async def exe():
    await asyncio.gather(
        asyncIteration('task1',2**(-21),2), # 2つの処理が
        asyncIteration('task2',2**(-20),4)  # 並行実行される形
    )

# 非同期処理の実行
print('開始')
t1 = time.time()
asyncio.run( exe() )
t2 = time.time()
print('終了:',t2-t1,'秒')
```

このプログラムは先の例と同じ処理を並行実行するものである．関数 long_task(n,v) は非同期処理のためのコルーチンとして定義されており，タスクの識別名 n と乱数の終了条件の値 v を引数に取る．このコルーチンでは，多数回反復される while の処理の中の

```
await asyncio.sleep(0)
```

において実行の管理をイベントループに移している．待ち時間を 0 にしているが，これは実行待ちをさせるのが目的ではなく，while による反復で毎回イベントループに制御を渡すのが目的である．すなわち，この行の記述がないと while ループが終了するまで当該スレッドが他のタスクの実行をブロックしてしまう．

コルーチン asyncIteration は時間のかかる long_task を指定した回数（引数 i）繰り返すものである．また，コルーチン exe はイベントループで実行する 2 つのタスクを起動するものである．

このプログラムを実行した際の出力の例を次に示す.

```
開始
task2 2.723247869429457e-07
task2 6.862664404527763e-07
task2 9.22241627954179e-07
task1 3.4535740955909944e-07
task2 5.80478085154823e-07
task1 2.7896666210391885e-07
終了:  7.798398494720459 秒
```

2 つのタスクが互いにブロックせずに実行されている様子がわかる.

4.5.6.1 非同期処理とイベントループの概観

複数のタスクの実行がイベントループ上で管理される様子を図 35 に示す.

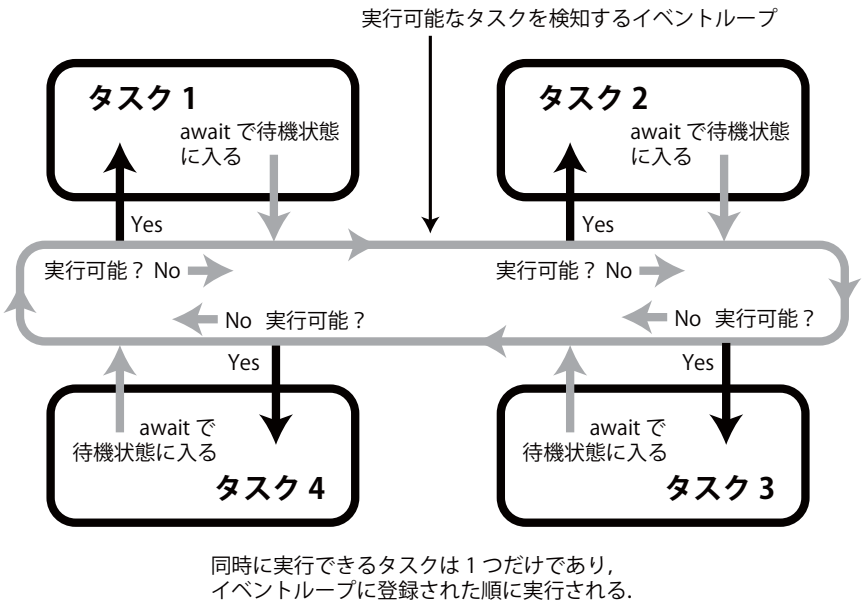

図 35: イベントループで監視されるタスクの概観

4.5.7 スレッドを非同期処理で管理する方法

イベントループに対する run_in_executor メソッドを使用すると，Awaitable でない通常の関数をメインのスレッドとは別のスレッドとして起動して非同期処理の枠組みで管理することができる．

書き方： イベントループ.run_in_executor(実行器, 関数, 引数並び…)

「関数」に「引数並び…」を与えて実行するための asyncio.Future オブジェクトを返す．この関数を実行する際に使用される「実行器」（concurrent.futures.ThreadPoolExecutor など）を与える．「実行器」として None を与えるとシステムが用意したものが採用される．

このメソッドで得られた Future オブジェクトをコルーチンにしたものをタスクに変換すると，イベントループ上の非同期処理として実行することができる．このことを次のサンプルプログラム asyncio05.py で解説する．

プログラム：asyncio05.py

```
# coding: utf-8
import asyncio, time
from concurrent import futures

def normalf(s):          # 通常の関数
    for i in range(3):
        print(s)
        time.sleep(1)
    return s

# Futureオブジェクトを実行するコルーチンの作成
async def fut2cor(f):
    c = await f
    return c

# スレッド実行器
e = futures.ThreadPoolExecutor(max_workers=2)

async def exe():    # タスクを投入して実行するコルーチン
    eLoop = asyncio.get_running_loop()  # イベントループを取得
    # 通常の関数のFutureオブジェクトを作成
    f1 = eLoop.run_in_executor(e, normalf, '通常の関数1')
    f2 = eLoop.run_in_executor(e, normalf, '通常の関数2')
    # Futureオブジェクトをタスクに変換
    t1 = asyncio.create_task(fut2cor(f1))
    t2 = asyncio.create_task(fut2cor(f2))
    # 両方のタスクを同時に実行
    r = await asyncio.gather(t1,t2)
    # 戻り値の確認
    print(r)
```

```
31      # 実行器の終了
32      e.shutdown()
33
34  asyncio.run(exe())  # イベントループの起動
```

解説）

このプログラムの関数 normalf は，引数に与えられた値を出力して 1 秒間待つ処理を 3 回繰り返すもので，これをイベントループ上で実行する．

run_in_executor はイベントループに対して実行するものなので，システムが用意したイベントループを 20 行目で取得している．22,23 行目で関数 normalf を実行するための Future オブジェクト f1，f2 を作成している．そして，それを関数 fut2cor でコルーチンに変換し，create_task 関数でタスク t1，t2 にしている．(25,26 行目)

このプログラムの実行結果を次に示す．

例. asyncio05.py をスクリプトとして実行

```
通常の関数 1
通常の関数 2
通常の関数 1
通常の関数 2
通常の関数 1
通常の関数 2
['通常の関数 1', '通常の関数 2']
```

スレッド上で実行される関数自体は非同期処理を意識せずに実装できるので，それを非同期処理の枠組みの上で管理したい場合にこの方法が有効である．

勿論のことではあるが，コルーチンをタスクにしたものと，スレッドをタスクにしたものを同時にイベントループ上で実行することができる．そのことを次の asyncio06.py で示す．

プログラム：asyncio06.py

```
1   # coding: utf-8
2   import asyncio, time
3   from concurrent import futures
4
5   def normalf(s):              # 通常の関数
6       for i in range(3):
7           print(s)
8           time.sleep(1)
9       return s
10
11  async def coroutine(s):      # コルーチン
12      for i in range(3):
13          print(s)
14          await asyncio.sleep(1)
15      return s
16
17  # Futureオブジェクトを実行するコルーチンの作成
18  async def fut2cor(f):
19      c = await f
20      return c
21
22  # スレッド実行器
23  e = futures.ThreadPoolExecutor(max_workers=1)
24
25  async def exe():    # タスクを投入して実行するコルーチン
26      eLoop = asyncio.get_running_loop()  # イベントループを取得
27      # 通常の関数のFutureオブジェクトを作成
28      f = eLoop.run_in_executor(e, normalf, '通常の関数')
29      # タスクを作成
30      t1 = asyncio.create_task(fut2cor(f))                # スレッドのタスク
31      t2 = asyncio.create_task(coroutine('コルーチン'))   # コルーチンのタスク
32      # 両方のタスクを同時に実行
33      r = await asyncio.gather(t1,t2)
34      # 戻り値の確認
35      print(r)
36      # 実行器の終了
37      e.shutdown()
38
```

```
39 asyncio.run(exe())  # イベントループの起動
```

このプログラムの実行結果を次に示す.

例. asyncio06.py をスクリプトとして実行

```
通常の関数
コルーチン
通常の関数
コルーチン
通常の関数
コルーチン
['通常の関数', 'コルーチン']
```

normalf と coroutine がタスクとして同時に実行されている様子がわかる.

4.5.8 非同期のコンソール入力を実現するためのライブラリ：aioconsole

有志のエンジニア Vincent Michel 氏が開発して公開しているライブラリ aioconsole [181] を用いると，input 関数と同様のコンソール入力を非同期に実行することができる．このライブラリは Python 言語処理系の標準ライブラリではなく，pip コマンドなどでシステムに導入する必要がある.

非同期のコンソール入力には ainput 関数を用いる.

書き方：　ainput(プロンプト)

「プロンプト」に与えた文字列をコンソールに表示して入力を促し，コンソールから入力された文字列を返す.

ainput 関数を使用したサンプル asyncio08.py を次に示す.

プログラム：asyncio08.py

```
 1 # coding: utf-8
 2 import asyncio
 3 import aioconsole
 4
 5 # 非同期のタスク1
 6 async def msg( s ):
 7     for i in range(3):
 8         print( s )
 9         await asyncio.sleep(4)
10     return s
11
12 # 非同期のタスク2
13 async def asyncInput():
14     while True:
15         r = await aioconsole.ainput('endで終了> ')
16         print(' 入力値:',r)
17         await asyncio.sleep(4)
18         if r == 'end': break
19
20 # タスク投入用コルーチン
21 async def exe():
22     g = await asyncio.gather(
23         msg('文字列表示'),      # タスク1
24         asyncInput()            # タスク2
25     )
26
27 asyncio.run( exe() )    # イベントループで実行
```

このサンプルでは 2 つのコルーチン msg と asyncInput が定義され，それらが非同期のタスクとして実行される．msg は引数に与えられた文字列を 3 回出力する.（毎回の出力の後で 4 秒待機する）asyncInput は非同期に実行され，他のタスクをブロックすることなくコンソールからの入力を受け付ける．このサンプルの実行例を次に示す.

[181] https://github.com/vxgmichel/aioconsole

例. asyncio08.py の実行

```
文字列表示                          ← msg による出力
end で終了> abcd   Enter            ←キーボードからの入力
 入力値:  abcd
文字列表示                          ← msg による出力
end で終了> end    Enter            ←キーボードからの入力
 入力値:  end
文字列表示                          ← msg による出力
```

msg と asyncInput が互いに相手をブロックせずに動作している様子がわかる.

4.5.9 永続するイベントループ上でのタスク管理

ここまでの解説では，イベントループ上のタスクが終了した時点で当該イベントループも終了する形の実装を示してきた．この形とは別に，永続するイベントループに対して動的にタスクを投入するという形式のプログラミング[182]も実際に多く行われている．ここでは，永続するイベントループとその上でのタスクの管理の方法について解説する.

4.5.9.1 永続するイベントループ

イベントループに対して run_forever メソッドを実行するとそのイベントループの動作が永続的になる.

書き方： イベントループ.run_forever( )

また，永続するイベントループを停止するには stop メソッドを使用する.

書き方： イベントループ.stop( )

動作を終了して停止したイベントループは close メソッドで閉じておくこと.

書き方： イベントループ.close( )

永続するイベントループに対して動的にタスクを投入する例を asyncio09.py で示す.

プログラム：asyncio09.py

```
 1 # coding: utf-8
 2 import asyncio
 3 import aioconsole
 4
 5 #--- タスクの定義 ----------------------------------------------------------
 6 # 非同期のタスク：メッセージを3回出力
 7 async def msg( n, s ):
 8     for i in range(3):
 9         print( '\t'*n + s )
10         await asyncio.sleep(1)
11     return s
12
13 # 非同期のタスク：コンソール入力
14 async def asyncInput():
15     while True:
16         r = await aioconsole.ainput('endで終了> ')
17         if r == 'end':
18             asyncio.get_running_loop().stop()   # イベントループの終了
19             print('終了します. ')
20             break
21         else:
22             print(' 入力値:',r)
23
24 # 最初に起動される非同期タスク
25 async def theFirst():
26     asyncio.create_task( msg(0,'タスク1') )      # 動的にタスクを投入(1)
27     await asyncio.sleep( 0.5 )
28     asyncio.create_task( msg(1,'タスク2') )      # 動的にタスクを投入(2)
29     await asyncio.sleep( 3 )
30     asyncio.create_task( asyncInput() )          # 動的にタスクを投入(3)
31
32 #--- イベントループの作成，最初のタスクの登録，永続実行 ----------------
```

[182] JavaScript 言語処理系（Web ブラウザのエンジンや Node.js など）におけるプログラミングもこのスタイルである.

```
33 eloop = asyncio.new_event_loop()     # イベントループを作成して
34 asyncio.set_event_loop( eloop )      # それを現行のスレッドにアタッチする.
35
36 eloop.create_task( theFirst() )      # 最初のタスクをイベントループに登録して
37 eloop.run_forever()                  # そのイベントループを永続実行する.
38
39 eloop.close()                        # 終了後はイベントループを閉じる.
```

このサンプルでは，メッセージの出力と 1 秒間の待機を 3 回繰り返すコルーチン msg と，コンソールからの非同期入力を受け付けるコルーチン asyncInput が定義されている．これらが，別のコルーチン theFirst から順次起動される形になっている．asyncInput の入力において 'end' と入力されるとイベントループが stop メソッドにより停止され，プログラム全体が終了する．

asyncio09.py を実行した際のコンソールの表示を次に示す．

例. ターミナルウィンドウで asyncio09.py を実行した様子

```
タスク 1
      タスク 2                ← 2 つのタスク msg が互いに
タスク 1
      タスク 2                  ブロックせずに実行されている
タスク 1
      タスク 2                ←ここで 2 つのタスク msg が終了した
end で終了> end  Enter        ←終了を指示
終了します.
```

参考）

このサンプルでは，イベントループ eloop を作成した後で，

```
asyncio.set_event_loop( eloop )
```

として現行のスレッドにそれをアタッチしているが，当サンプルではこの行を省略することができる．

4.5.9.2 タスクの一覧取得とタスクのキャンセル

all_tasks 関数を用いると，イベントループの上で実行されているタスクの一覧（Set オブジェクト）を取得することができる．

書き方： asyncio.all_tasks(イベントループ)

「イベントループ」（ProactorEventLoop オブジェクト）の上で実行されているタスク（Task オブジェクト）を要素とする Set オブジェクトを返す．

実行中のタスクを終了する（キャンセルする）には cancel メソッドを使用する．

書き方： タスク.cancel()

「タスク」（Task オブジェクト）を強制終了する．この際に警告が発生することがあるのでそれを適切にハンドリングすること．

次に示すサンプル asyncio010.py は，コンソールからの非同期の入力を受けて新規のタスクをイベントループに随時投入するものである．

プログラム： asyncio10.py

```
 1 # coding: utf-8
 2 import asyncio
 3 import aioconsole
 4
 5 #--- タスクの定義 ----------------------------------------------------------
 6 # 非同期のタスク：無制限の繰り返し
 7 async def tsk():
 8     while True: await asyncio.sleep(1)
 9
10 # 非同期のタスク：コンソール入力
11 async def asyncInput():
12     el = asyncio.get_running_loop()
13     while True:
14         r = await aioconsole.ainput('endで終了> ')
15         if r == 'end':
```

```
16              for t in asyncio.all_tasks( el ):
17                  t.cancel()                          # タスクを強制的にキャンセル
18                  try:                                # その際の例外処理
19                      await t
20                  except asyncio.CancelledError:
21                      pass                  # 必要に応じてここで例外処理を実行する
22              asyncio.get_running_loop().stop()   # イベントループの終了
23              print('終了します. ')
24              break
25          elif r == 'show':
26              print( '実行中>\t', end='' )
27              for t in asyncio.all_tasks( el ):           # 実行中タスクの
28                  print( t.get_name(), sep='', end='\t' ) # 一覧出呂っく
29              print()
30          elif r == 'new':
31              asyncio.create_task( tsk() )    # 動的にタスクを投入
32          else:
33              print(' 入力値:',r)
34
35  # 最初に起動される非同期タスク
36  async def theFirst():
37      asyncio.create_task( asyncInput() )
38
39  #--- イベントループの作成, 最初のタスクの登録, 永続実行 ----------------
40  eloop = asyncio.new_event_loop()    # イベントループを作成して
41
42  eloop.create_task( theFirst() )     # 最初のタスクをイベントループに登録して
43  eloop.run_forever()                 # そのイベントループを永続実行する.
44
45  eloop.close()                       # 終了後はイベントループを閉じる.
```

このサンプルでは，コンソールから 'show' と入力すると実行中のタスクの一覧（タスク名の一覧）を表示する．この際に all_tasks メソッドが使用される．また，タスク名の取得には当該タスクに対して get_name を実行する．

コンソールから 'new' と入力するとコルーチン tsk をタスクとしてイベントループに追加し，'end' と入力すると実行中のタスクを全て強制終了してプログラム全体を終了する．

実行中のタスクを cancel メソッドで強制的に終了するとしばしば警告（asyncio.CancelledError）が発生するので，これを適切にハンドリングすること．

asyncio10.py を実行した際のコンソールの表示を次に示す．

例. ターミナルウィンドウで asyncio10.py を実行した様子

```
end で終了> show   Enter        ←状態確認
実行中> Task-2                  ←この段階では asyncInput だけが実行されている
end で終了> new    Enter        ←新規タスク投入
end で終了> show   Enter        ←状態確認
実行中> Task-5 Task-2           ←タスクが 1 つ増えている
end で終了> new    Enter        ←新規タスク投入
end で終了> show   Enter        ←状態確認
実行中> Task-5 Task-8 Task-2    ←タスクが 1 つ増えている
end で終了> end    Enter        ←終了を指示
終了します.
```

■ タスクの名前について

イベントループに投入されたタスクには自動的に名前が付与される．先の実行例で得られた

```
Task-5 Task-8 Task-2
```

のような出力には各タスクに自動的に命名されたものが見られる．タスク投入時に意図して名前を与える場合は，キーワード引数 'name=' にそれを指定して，

```
asyncio.create_task( コルーチン, name=タスク名 )
```

のように記述する．

4.5.10　非同期のイテレーション

for 文による反復処理に async キーワードを付けることで**非同期のイテレーション**を実現することができる.

書き方：　async for 変数 in 非同期のイテラブル:
（反復対象の処理）

「非同期のイテラブル」（後述）から要素を順次「変数」に受け取って「反復対象の処理」を実行する.「変数」に値を受け取る際に await による待機が起こる場合，イベントループに実行の制御が移される.

4.5.10.1　非同期のイテラブル

要素の取得（あるいは生成）に時間がかかる（await による待機が起こる）イテラブルは**非同期のイテラブル**[183] として扱うことができる．非同期のイテラブルには様々なものがあり，ファイル入出力や通信における**入力ストリーム**を非同期の形で扱うイテラブルが代表的な例である．また，次に説明する**非同期のジェネレータ**，**非同期のイテレータ**もこの範疇に入る.

■ 非同期のジェネレータ

非同期のジェネレータは async キーワードで宣言された**ジェネレータ**[184] である．非同期のジェネレータの例を次のサンプル asyncio11.py で示す.

プログラム：asyncio11.py

```
1  # coding: utf-8
2  import asyncio
3
4  # コルーチン1
5  async def msg():
6      for x in range(4):
7          await asyncio.sleep(1)
8          print( 'タスク1' )
9
10 # コルーチン2：非同期のジェネレータ
11 async def aGen():
12     for x in ['a','b']:
13         await asyncio.sleep(2)
14         yield x
15
16 # タスクを投入するコルーチン
17 async def exe():
18     asyncio.create_task( msg() )     # タスク1
19     async for x in aGen():           # 非同期の反復処理
20         print( '\t非同期に値を取得:',x )
21
22 asyncio.run( exe() )     # イベントループで実行
```

このサンプルでは非同期のジェネレータ aGen がコルーチンの形で定義されている．このジェネレータは 2 つの要素 'a'，'b' を順次返すものであり，値を返す前に 2 秒の待ち時間がある．また，これとは別にコルーチン msg が定義されており，1 秒の待ち時間の後でメッセージを出力することを 4 回繰り返す.

aGen は非同期のイテレーション（async for 文）で使用され，要素取得までの待ち時間の間も他のタスク msg をブロックしない.

このサンプルを実行した際の出力の例を次に示す.

例. ターミナルウィンドウで asyncio11.py を実行した様子

```
タスク1                          ←msg による出力
        非同期に値を取得:  a      ←async for 文内の print による出力
タスク1                          ←msg による出力
タスク1                          ←msg による出力
        非同期に値を取得:  b      ←async for 文内の print による出力
タスク1                          ←msg による出力
```

[183] **非同期のコンテナ**と呼ぶこともある.

[184] 後の「4.7 ジェネレータ」（p.279）で詳しく解説する.

msg のタスクと非同期のイテレーションの処理が互いにブロックされずに実行されている様子がわかる．

■ 非同期のイテレータ

非同期のイテレータとは，要素を返す処理を非同期に行うイテレータである．基本的な実装方法は先の「2.9.6 イテレータのクラスを実装する方法」（p.154）に似ており，2 つのメソッド __aiter__ ，__anext__ の実装による．特に __anext__ メソッドは値を返す処理を非同期に実行するので async キーワードを付けて宣言する．

非同期のイテレータの例を次のサンプル asyncio12.py で示す．これは先の asyncio11.py と同等の処理を実行する（先の実行例と同様の出力を得る）ものである．

プログラム：asyncio12.py

```
# coding: utf-8
import asyncio

# コルーチン1
async def msg():
    for x in range(4):
        await asyncio.sleep(1)
        print( 'タスク1' )

# 非同期のイテレータのクラス
class AsyncIter:
    def __init__(self):             # コンストラクタ
        self.Q = ['a','b']
        self.cur = 0
    def __aiter__(self):
        return self
    async def __anext__(self):      # 次の要素を返すメソッド（非同期処理）
        if self.cur > 1:                # イテレーションが終了している場合に
            raise StopAsyncIteration    # この例外を起こす
        else:
            await asyncio.sleep(2)
            r = self.Q[self.cur]
            self.cur += 1
            return r

# タスクを投入するコルーチン
async def exe():
    asyncio.create_task( msg() )    # タスク1
    A = AsyncIter()                 # 非同期のイテレータの作成
    async for x in A:               # 非同期の反復処理
        print( '\t非同期に値を取得:',x )

asyncio.run( exe() )    # イベントループで実行
```

このサンプルでは非同期のイテレータのクラス AsyncIter が定義されている．コルーチン exe の中でこのクラスのインスタンス A が作成されて非同期のイテレーションに使用されている．

4.5.11　非同期のファイル入出力：aiofiles

aiofiles ライブラリは，asyncio ライブラリと連携して非同期のファイル操作を行う機能を提供する．Python 処理系の通常のファイル操作は同期処理であり，処理が完了するまで当該スレッドをブロックする．aiofiles はこれを回避するために，ファイル操作を別のスレッドプールに委譲する形で非同期のファイル操作を行う．

aiofiles ライブラリは Python の標準ライブラリではなく，サードパーティ[185] が提供するものであり，これを使用するには pip コマンドなどで Python 言語処理系の環境に予め導入しておく必要がある．

aiofiles ライブラリが提供する API は，Python の標準的なファイル操作用の API の場合と同様の記述ができるように配慮されている．また，標準ライブラリである os モジュールが提供するファイル操作関連の API と同じ名称の API が多数提供されており，os モジュールの場合と同様の記述方法で使用できる点も大きな特徴である．

[185] Tin Tvrtkovic 氏によって開発された．（https://pypi.org/project/aiofiles/）

aiofiles によるファイル入出力は「2.7 入出力」(p.94) で解説した API と類似の用法で実現できる．すなわち，open 関数でファイルを開いてファイルオブジェクトを作成し，それに対して read，write メソッドを実行して入力，出力を行い，最後に close メソッドを実行してファイルを閉じる．(それらの関数やメソッドを await 式として実行する)

本書では使用例を挙げて aiofiles の最も基本的な API について解説する．提供されている各 API とその詳細に関しては aiofiles の公式インターネットサイトなどを参照のこと．

4.5.11.1 サンプルプログラム

次に示す aiofiles01.py では，2 つのタスク aFileIOr2()，msg() がイベントループ上で互いにブロックすることなく実行される．aFileIOr2() のタスクは，サイズの大きなファイルを作成して直後にそれを読み込むことを 2 回繰り返すというもので，入出力に伴う待ち時間が発生するものである．また msg() のタスクは一定の時間間隔でメッセージを繰り返しコンソールに出力するもので，1 つ目のタスクが実行を終えた後で終了する．

プログラム：aiofiles01.py

```
# coding: utf-8
import asyncio
import aiofiles

flg = True      # 共有変数：aFileIOr2 が実行中はTrue，終われば False

# ファイル出力と読込みを2回繰り返す処理
async def aFileIOr2():
    global flg
    for _ in range(2):
        print('ファイルの作成')
        # ファイルの作成と出力
        f = await aiofiles.open('aiofiles01.bin','w')
        for n in range(50000):      # 50,000回出力（反復回数は適宜調整する）
            c = chr( n%26 + 97 )    # a～zの文字を作成
            await f.write(c)        # 1文字ずつ出力
            if c == 'z': await f.write('\n')
        await f.close()
        print('ファイルの読込み')
        # 上で作成したファイルの読込み
        f = await aiofiles.open('aiofiles01.bin','r')
        while True:
            c = await f.read(1)
            if not c: break
        await f.close()
    flg = False

# 定期的にメッセージを出力する処理
async def msg():
    global flg
    while flg:      # aFileIOr2()が終了するまで繰り返す
        await asyncio.sleep(2.5)    # 待ち時間は適宜調整する
        print('\t\t別のタスク')

# タスク投入用コルーチン
async def exe():
    t1 = asyncio.create_task( aFileIOr2() )
    t2 = asyncio.create_task( msg() )
    await t1;   await t2

asyncio.run( exe() )    # 実行
```

このプログラムの実行例を次に示す．

例. aiofiles01.py を OS のターミナルウィンドウで実行した際の出力

```
ファイルの作成
                別のタスク
ファイルの読込み
                別のタスク
ファイルの作成
                別のタスク
ファイルの読込み
                別のタスク
                別のタスク
```

このように，2 つのタスク aFileIOr2()，msg() がイベントループ上で互いにブロックすることなく実行されているのがわかる．

このプログラムの実行によって作成されるファイル aiofiles01.bin は次のような内容となる．

```
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
        ⋮
（以下同様）
        ⋮
```

上記プログラム中ではファイルを開く処理に open 関数を使っており，引数の仕様も標準の open 関数に準じている．ただし，aiofiles の open 関数が返すファイルオブジェクトのクラスは表 35 に示すように，標準の open 関数が返すものとは異なる．

表 35: open 関数が返すファイルオブジェクトのクラス（一部）

	テキスト	バイナリ
入力用	AsyncTextIOWrapper	AsyncBufferedReader
出力用	（同上）	AsyncBufferedIOBase

ここで示したサンプル aiofiles01.py は，ファイルの入出力の待ち時間を強調するために，敢えて 1 バイトずつ入出力を行ったが，実用的な局面では長いデータを適宜まとめて（あるいは全ての内容を一度に）入出力することになる．

ファイル入出力を頻繁に実行しながら他の処理を並行して実行することが求められるアプリケーションを作成する場合に，この aiofiles が役立つ．

■ 非同期処理の通信への応用

asyncio による非同期処理の通信への応用に関しては，後の「5 TCP/IP による通信」（p.338）の中の「5.1.6 実用的な通信機能を実現する方法」（p.341）で解説する．

参考）

サードパーティが開発して公開している uvloop ライブラリを用いると高速なイベントループが実現できる．これは，JavaScript 処理系の V8 エンジンで採用されているイベントループ機能（libuv）を Python 向けに応用したものである．